<알고리즘 Mix Tape 04>

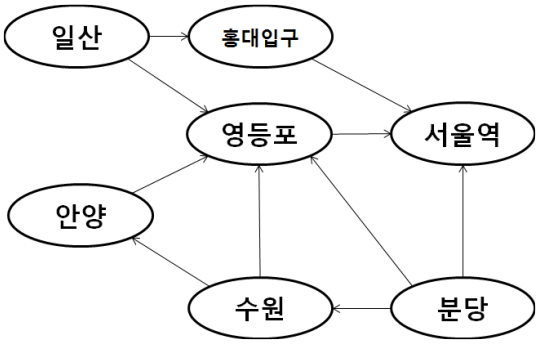
8장_3 위상정렬과 다익스트라 알고리즘

(170605 to 170607)

우리가 대학교 때 선수 과목이란 개념 때문에 시간표를 어떻게 들어야 될까에 대한 고민을 해본 적이 있을 것이다. 또한 어렸을 때 드라마 야인시대를 본 사람이 있는지 모르겠는데 등장인물의 우열을 판단하기 애매한 경우가 있을 것이다. 이러한 경우에 쓰이는 알고리즘이 바로 위상 정렬(Topological Sort)를 이용해서 표현할 수 있다. 그리고 인접한 정점과 가까운 가중치들을 탐색해보면서 Prim 알고리즘과 융합해서 최소의 경로를 찾는 알고리즘인 다익스트라 알고리즘이 있다. 이번 요약정리에서는 방금 무향 그래프와는 달리 유향(有向) 그래프에서 쓰이는 그래프 알고리즘에 대해서 자세히 공부해보도록 하자.

1. 위상 정렬(Topological Sort)

우리가 세상을 살아가면서 진로를 결정하는 경우가 있기 마련이다. 그렇지만 그 분야에 대해서 접근을 하는 방법이 있다면 분명 무언가를 또 공부를 해야 할 것이다. 인생의 복잡한 진로들을 해결하는 방안대로 순서를 매기는 방법이 있듯이 정점들로 구성되며 간선이 방향성이 존재하여 정점들의 절차적인 순서를 꼬이지 않도록 정리하는 원리를 위상 정렬이라고 한다. 예를 들어 서울역에 통근을 하는 경우가 있는데 서울역에 주거 지역이 따로 없으니 베드타운 지역(예를 들어 경기도 일대가 될 수 있겠다.)에서 서울역까지 접근을 하는데 있어서 간단한 사례로 아래와 같이 정리해볼 수 있다.



그래프를 딱 봐도 생각보다 복잡할 것으로 판단을 할 것이다. 쉽게 이야기해서 서울역으로 통근하는 지역들은 대개 분당이든 수원이든 일산이든 그 지역들로부터 시작을 하지만 위상 정렬의 순서는 인덱스의 순서에 따라 달라질 수밖에 없다. 위 그래프의 정의를 위하여 아래 코드를 참고하면서 알아보도록 하자.

```
static Vertex[] V = new Vertex[] {
 new Vertex("일산"), new Vertex("홍대입구"), new
Vertex("영등포"),
 new Vertex("서울역"), new Vertex("안양"), new
Vertex("수원"), new Vertex("분당")
};
static Edge[] E = new Edge[] {
 new Edge(V[0], V[1]), new Edge(V[0], V[2]), new
Edge(V[1], V[3]),
 new Edge(V[2], V[3]), new Edge(V[4], V[2]), new
Edge(V[5], V[2]), new Edge(V[5], V[4]), new Edge(V[6],
V[2]), new Edge(V[6], V[3]), new Edge(V[6], V[5])
};
```

```
static Vertex[] topologicalSort() {
  Vertex[] A = new Vertex[V.length];
  for (int i = 0; i < A.length; ++i) {
    for (int j = 0; j < V.length; ++j)
      if (V[j] != null && V[j].parentCount == 0) {
       A[i] = V[j]; // 1
        for (int k = 0; k < E.length; ++k)
         if (E[k] != null && E[k].parent == V[j]) {
            E[k].child.parentCount--;
            E[k] = null;
          } // 2
          V[j] = null; // 3
          break;
      }
    }
    return A;
}
public static void main(String[] args) {
    Vertex[] a = topologicalSort();
    for (Vertex v : a)
      System.out.println(v.title);
}
```

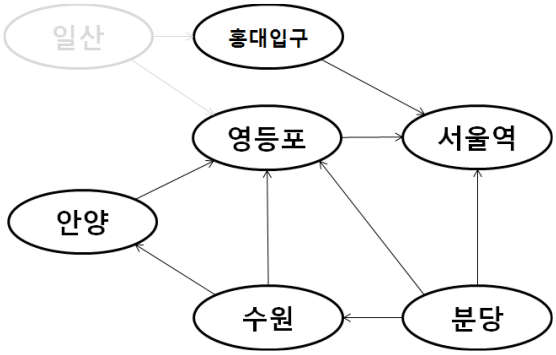
여기서는 일산을 인덱스 0, 홍대입구를 인덱스 1, 영등포를 인덱스 2, 서울역을 인덱스 3... 이렇게 한 것을 알 수 있다. 위상 정렬에서 가장 영향을 주는 것이 바로 유향 그래프를 구성하는 정점의 인덱스이다. 위상 정렬 알고리즘에서도 실제로는 정점의 순서대로 살펴보면서 진입 간선과 정점을 지우는 원리로 볼 수 있기 때문이다. 이러한 알고리즘이 어떻게 돌아가는지에 대해 그림과 알고리즘의 내용을 통해서 공부를 해보도록 하자.

0단계) 정점의 진입 차수에 대해 정리해두자.
여기서 진입 차수는 parentCount로 작성할 수 있다. 각 정점들 별로 진입 차수를 분별해서 진입 차수가 0인 정점들에 대해 차례대로 삭제를 하는 것이 곧 위상 정렬의 과정이다.

일산	홍대	영등포	서울역	안양	수원	분당
0	1	4	3	1	1	0

여기서 A는 위상 정렬이 완료된 정점들에 대해서 차례대로 저장을 하여 반환을 하는 정점이다. 이제 A에 대한 반복문을 돌아보면서 위상 정렬을 완성해보자.

1단계) A의 0번째 인덱스는 누가 채울까?
일단 인덱스 순서대로 살펴보면 일산의 인덱스가 0인 것을 발견할 수 있다. 그러면 배열 A에 일산 정점을 추가하는 대신에 실제 그래프의 정점에서는 없애버리고 일산의 진출 간선에 대해 없애버린다.

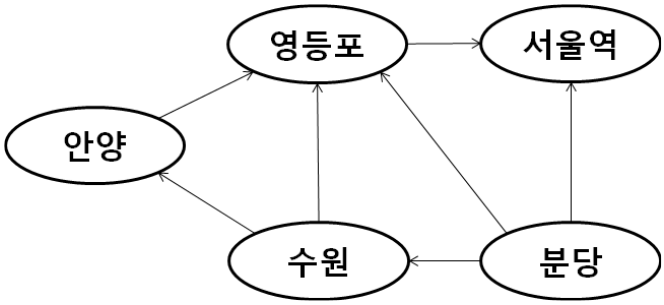


그러면 현재 진입 차수들에 대하여 정리를 한다면 아래와 같이 정리된다.
-> 2번 문장 이후

~~일산~~	홍대	영등포	서울역	안양	수원	분당
~~0~~	0	3	3	1	1	0

-> 3번 문장 이후로는 정점 일산의 모든 내용이 null로 된다.
위상 정렬 완료된 정점 : 일산

2단계) A의 1번째 인덱스를 결정하자
일단 정점 일산에 대해서는 이미 null 값이 되어 버렸기 때문에 제하고 시작해야 한다. 그럼 인덱스를 기준으로 진입 차수가 0인 정점은 바로 홍대입구이다. 그리하여 배열 A의 첫 번째 인덱스에는 홍대입구를 추가하는 대신에 홍대입구의 진출 간선에 대해 제한다.

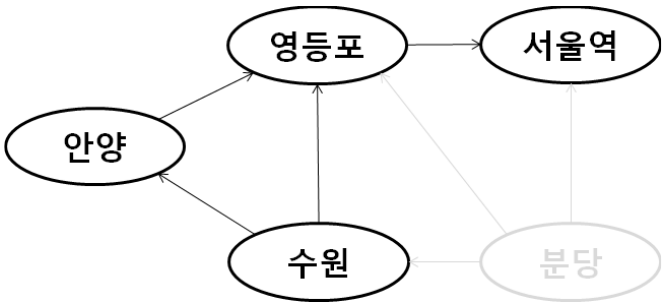


그러면 현재 진입 차수들에 대하여 정리를 한다면 아래와 같이 정리된다.
-> 2번 문장 이후

~~일산~~	~~홍대~~	영등포	서울역	안양	수원	분당
~~0~~	~~0~~	3	2	1	1	0

-> 3번 문장 이후로는 정점 홍대입구의 모든 내용이 null로 된다.
위상 정렬 완료된 정점 : 일산, 홍대입구

3단계) A의 2번째 인덱스를 결정하자
그러면 현재 일산과 홍대입구가 없어졌기 때문에 진입 차수가 없는 녀석은 바로 분당이다. 어자피 break 문이 있는 것으로 봐서 모든 정점들을 인덱스 순서대로 돌면서 진입 정점이 0인 것들을 찾기 때문에 인덱스 순서가 중요하다는 의미를 생각할 수 있을 것이다. 그럼 A의 두 번째 인덱스에는 분당이 추가된다.

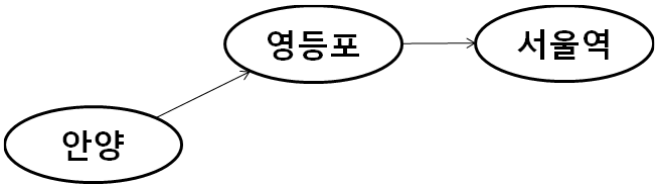


-> 2번 문장 이후

~~일산~~	~~홍대~~	영등포	서울역	안양	수원	~~분당~~
~~0~~	~~0~~	2	1	1	0	~~0~~

-> 3번 문장 이후로는 정점 분당의 모든 내용이 null로 된다.
위상 정렬 완료된 정점 : 일산, 홍대입구, 분당

4단계) A의 3번째 인덱스를 결정하자
위 과정을 마치다보면 진입 차수가 없는 정점은 바로 수원이다. 아까와 마찬가지로 수원은 이미 없는 정점이 되어 버리고 위상 정렬 목록에 올라가게 된다.

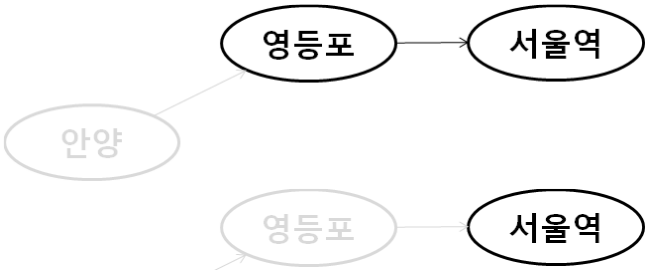


-> 2번 문장 이후

~~일산~~	~~홍대~~	영등포	서울역	안양	~~수원~~	~~분당~~
~~0~~	~~0~~	1	1	0	~~0~~	~~0~~

-> 3번 문장 이후로는 정점 수원의 모든 내용이 null로 된다.
위상 정렬 완료된 정점 : 일산, 홍대입구, 분당, 수원

5단계~7단계) A의 4~6번째 인덱스를 결정하자
위 과정에서 진입 차수가 없는 정점이 바로 안양이다. 이제 안양부터는 영등포랑 서울역이 남아 있기 때문에 각각 진출 간선과 정점들이 없어지면서 위상 정렬 목록에 추가된다.



-> 5, 6단계 각각 2번 문장 이후

~~일산~~	~~홍대~~	영등포	서울역	~~안양~~	~~수원~~	~~분당~~
~~0~~	~~0~~	0	1	~~0~~	~~0~~	~~0~~

~~일산~~	~~홍대~~	~~영등포~~	서울역	~~안양~~	~~수원~~	~~분당~~
~~0~~	~~0~~	~~0~~	0	~~0~~	~~0~~	~~0~~

-> 3번 문장 이후로는 각 단계 순서대로 정점 안양, 영등포의 모든 내용이 null로 된다.
-> 최종적으로 서울역의 정점까지 위상 정렬이 완료되면 아래와 같은 내용으로 정렬이 된다.
위상 정렬 완료된 정점 : 일산, 홍대입구, 분당, 수원, 안양, 영등포, 서울역

어쩌다 보면 위상 정렬은 진입 차수가 없는 간선들로부터 단계 별로 정렬하는 모습을 보면 위상 정렬도 크게 어렵게는 느껴지지 않는다. 하지만 위상 정렬의 시간 복잡도에 대해서 고려를 해보면 이 문장으로는 O(V*E)로 나와서 꽤 복잡한 시간 복잡도로 나왔다. 하지만 같은 시간 복잡도 범위 내에서 그나마 효율적으로 표현하는 방법에는 뭐가 있다고 생각이 드는가? 그렇다. 바로 인접 리스트를 이용하는 것이다.

```
static Vertex[] createGraph() {
    Vertex[] V = new Vertex[] {
  new Vertex("일산"), new Vertex("홍대입구"), new
Vertex("영등포"), new Vertex("서울역"), new
Vertex("안양"), new Vertex("수원"), new Vertex("분당")
```

```
        };
        V[0].addChildren(V[1], V[2]);
        V[1].addChildren(V[3]);
        V[2].addChildren(V[3]);
        V[4].addChildren(V[2]);
        V[5].addChildren(V[2], V[4]);
        V[6].addChildren(V[2]. V[3], V[5]);
        return V;
}
 static class Vertex {
    String title;
    int parentCount = 0;
    Vertex[] children = new Vertex[0];

    public Vertex(String title) {
        this.title = title;
    }

    public void addChildren(Vertex ... children)
{ // *
        this.children = children;
        for (Vertex child : children)
            child.parentCount++;
    }
}
```

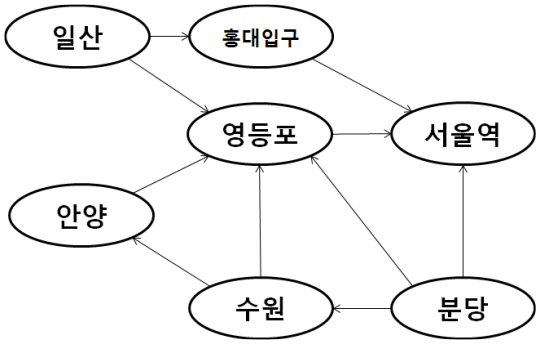
주목을 해야 하는 부분이 바로 *이다. 여기서 ...은 자바에서 아시다시피 배열을 넣을 수 있다는 사실을 알고 있을 것이다. 물론 C언어에서도 포인터를 통해서 배열을 넣는 경우도 부지기수한데 이러한 원리로 생각을 하면 되겠다. 그렇지만 이러한 기능을 실제로 추가한 이유가 아까 반복문은 너무나 복잡해서 각 간선에 대해 없애주는 작업까지 함께 하니 배로 귀찮았다. 하지만 이를 통해 작업하면 어느 한 정점에 대해서 진입 차수가 0인 정점들에 대해서 그 진출하는 정점들에 대해 진입 차수를 줄여주면 되기 때문에 같은 시간 복잡도에도 불구하고 그나마 효율적으로 표현한 방법이 바로 인접 리스트를 이용한 사례이다. 그렇지만 시간 복잡도도 줄이면서 가장 짧은 코드로 마무리를 할 수 있는 방법이 있는데 바로 8장_1에서 배웠던 DFS(Depth 1st Search)를 이용하는 것이다.

2. 위상정렬 X DFS

방금 DFS(Depth 1st Search, 깊이 우선 탐색)을 통해서 정점의 깊이를 통해 탐색하는 과정을 공부했을 것이다. 그렇지만 위상정렬을 효율적으로 하는 방안으로서는 깊이 우선 탐색을 통해서 깊이가 가장 깊은 정점들을 잡아서 정렬을 하는 원리로 생각할 수 있다. 깊이 우선 탐색의 시간 복잡도는 인접 리스트만 잘 사용하면 O(|V|+|E|)로 나오기 때문에 방금 위상 정렬 시간 복잡도보다 효율적으로 표현할 수 있다. 방금 살펴본 코드에 DFS를 이용해서 위상 정렬을 하는 원리에 대해서 공부를 해보도록 하자.

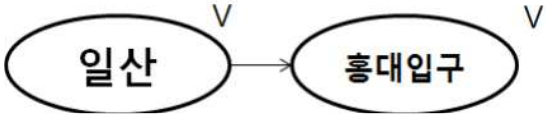
```
static Vertex[] createGraph() {
    Vertex[] V = new Vertex[] {
    new Vertex("일산"), new Vertex("홍대입구"), new
Vertex("영등포"), new Vertex("서울역"), new
Vertex("안양"), new Vertex("수원"), new Vertex("분당")
    };
    V[0].addChildren(V[1], V[2]);
    V[1].addChildren(V[3]);
    V[2].addChildren(V[3]);
    V[4].addChildren(V[2]);
    V[5].addChildren(V[2], V[4]);
    V[6].addChildren(V[2], V[3], V[5]);
    return V;
}
static LinkedList<Vertex> topologicalSort(Vertex[] V) {
    LinkedList<Vertex> A = new LinkedList<>();
    HashSet<Vertex> visited = new HashSet<>();
    for (Vertex v : V) // *
      if (visited.contains(v) == false) DFS_TS(v, visited,
A);
      return A;
}

static void DFS_TS(Vertex v, HashSet<Vertex> visited,
LinkedList<Vertex> A) {
    visited.add(v);
    for (Vertex child : v.children)
      if (visited.contains(child) == false) DFS_TS(child,
visited, A);
      A.addFirst(v); // +
}
```

방금 위상 정렬 알고리즘과의 차이점은 무엇일까? 여기서는 진입 차수는 있으면서 진출 차수가 없는 정점들에 대해서 깊이가 가장 깊은 정점으로 생각을 하고 정점 목록에 저장을 하는 것이다. 깊이가 가장 깊은 정점에 대하여 탐색을 해서 목록에 추가를 해서 이용하는 것이기 때문이다. 그래서 이번에도 어떠한 원리로 돌아가는지에 대해 숙지해보도록 하자.

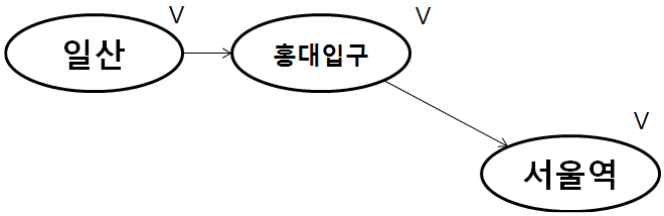


1단계) *에서 V가 0번째(일산)인 경우
우선 일산으로 시작을 하게 되면 홍대입구, 영등포를 방문을 하고 난 후, 그 두 역에 대해서 최종적으로 서울역을 방문하는 시나리오로 끝나게 된다.

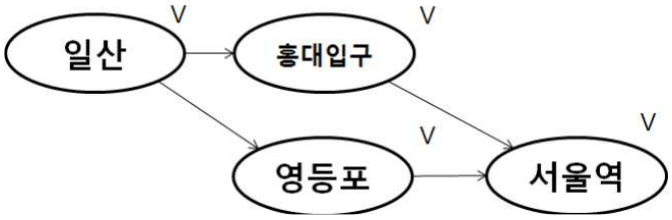
2단계) 일산 근처에 있는 정점들을 방문처리하기
그럼 일산으로 진출하는 정점들 중에서 인덱스가 가장 낮은 인덱스에 대해 순차적으로 방문을 하니 홍대입구(인덱스 1)에 대해서 DFS 탐색을 하게 된다.



그리고 일산과 홍대입구는 방문 처리가 되었기 때문에 이제 홍대입구의 진출 간선을 통하여 깊이가 깊은 정점인 서울역(인덱스 3)에 대해 방문을 하게 된다.

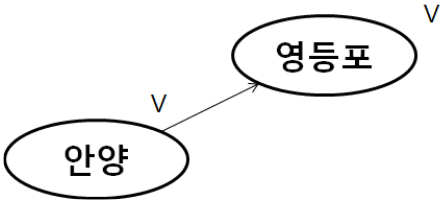


그럼 일단 서울역이 + 문장을 통해서 DFS 위상 정렬 결과에 맨 첫 번째에 올라오게 된다. 아마 DFS는 Stack의 원리를 통해서 작동한다고 생각을 할 수 있는데 위상 정렬에서는 깊이가 깊은 정점에 대해서 pop() 연산을 하는 것으로 볼 수 있겠다. 그리고 방금 방문했던 홍대입구에 대해서도 + 문장을 통해서 맨 앞에 나오게 되어 결국 홍대입구, 서울역으로 정렬이 되었다. 그러나 여기서 방문을 안 한 정점인 영등포가 있다. 영등포에 대해서 방문 처리를 하자.



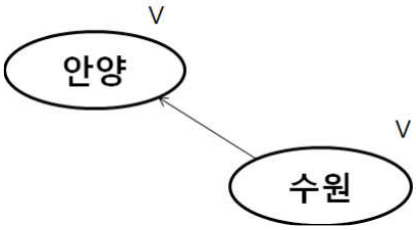
그럼 결국 영등포도 방문 처리가 되었음으로 + 문장을 통해서 (영등포, 홍대입구), 일산으로 저장이 된다. 그럼 일산에 인접한 정점들에 대하여 모든 방문이 완료되었으니 일산에 대해서도 + 문장을 통해서 최종적으로 일산, (영등포, 홍대입구), 서울역으로 상황이 종료된다.

3단계) 일산에서 서울역까지 모두 방문을 했으니 * 문장을 통해 4번째 인덱스인 안양을 방문하자



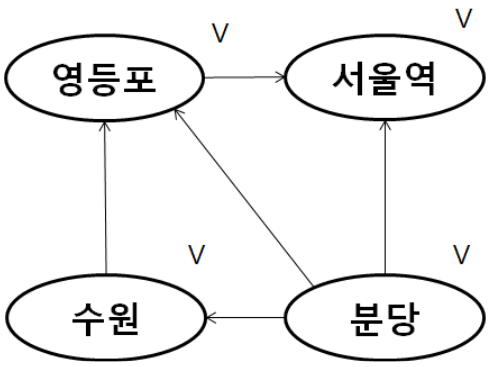
안양으로 진출하는 간선은 영등포에 도달을 한다. 그러나 영등포는 이미 방문을 하였기 때문에 안양에 대해서 + 문장을 통해 추가를 하면 안양, 일산, (영등포, 홍대입구), 서울역으로 정리할 수 있다.

4단계) * 문장을 통해 5번째 인덱스인 수원을 방문하자

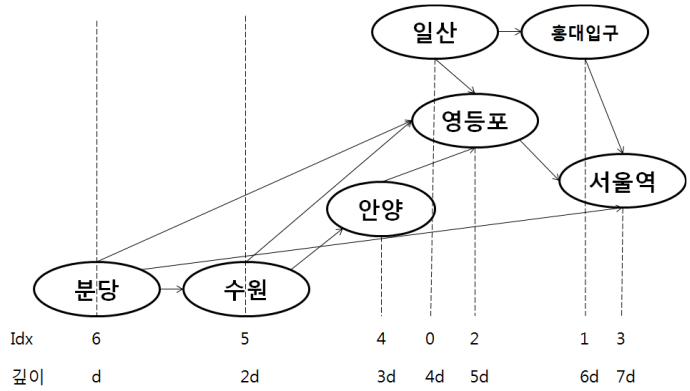


수원과 인접한 정점인 안양은 이미 방문한 셈이다. 그래서 + 문장을 통해 추가를 하면 수원, 안양, 일산, (영등포, 홍대입구), 서울역으로 정리가 가능하다.

5단계) * 문장을 통해 마지막 정점인 분당을 방문하자



이미 분당과 인접한 모든 정점들은 최종적으로 방문을 마친 상태이다. 그래서 + 문장을 통해 맨 앞에 분당을 넣으면 최종적으로 분당, 수원, 안양, 일산, 영등포, 홍대입구, 서울역으로 정렬이 된다. 그렇지만 아까 1번째 결과랑 어지간히 다른 결과가 나오게 된다. 첫 번째는 오로지 인덱스 순서를 통해서 순차적으로 탐색을 하였기 때문에 이런 결과가 나왔지만, 실제로 아까 우리가 이용했던 그래프를 변형해서 그려보도록 하자.



우리가 유향 그래프에서는 깊이가 깊다 라는 의미를 받아들이기 위해서는 인덱스가 다가 아니라는 상식을 깨야 한다. 이처럼 그래프를 일부러 그린 이유는 인덱스랑 깊이에 대해서 가상적으로 차이를 느껴봄으로서 비록 인덱스가 크게 느껴진다고 하더라도 진입된 깊이에 따라서 달라질 수도 있다는 의미로 받아들일 수 있다는 뜻이다. 물론 정점은 어디에서 시작하는지에 따라 결과는 달라질 수도 있다. 시작된 정점을 기반으로 상대적인 깊이가 달라지기 때문이다.

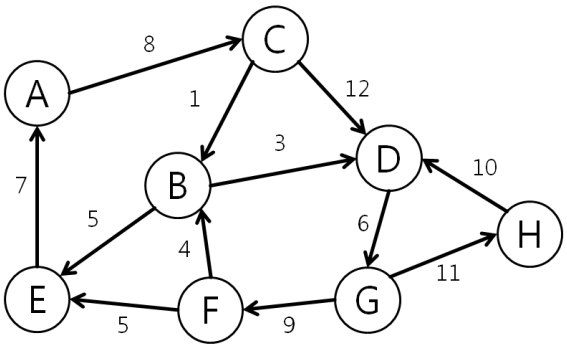
(참고) DAG 그래프??
여기서 DAG를 잠깐 언급하겠다. Directed Acyclic Graph로서 여기서 Acyclic이란 뜻은 '순환, 그런 거 없다.' 라는 뜻이다. 쉽게 말해서 '유향 그래프 속에 순환이 존재하지만 않으면 된다.' 라는 뜻이다. 솔직히 유향 그래프에 순환이 있다면 위상 정렬을 할 수가 없게 된다. 쉽게 이야기하면 같은 내용을 반복해 이야기 하는 사람이 있으면 토론이 끝나지 않아서 짜증난다고 느낄 수 있는 거와 똑같다. 위상 정렬을 하기 위한 유향 그래프는 반드시 DAG 그래프를 형성하길 바란다.

3. 다익스트라(Dijkstra) 알고리즘

아마 운영체제론 시간에 멀티 프로그래밍 환경에서 프로세스들의 교착상태(Deadlock)를 다루기 위해서

쓰인 세마포어(Semaphore) 개념을 발표한 양반이 바로 다익스트라이다. 실제로 다익스트라는 그래프에 대해 꼴렸는지 유향 가중치 그래프를 통해서 최소의 가중치를 통한 경로를 찾아내기 위한 알고리즘을 발표했는데 바로 이 알고리즘이 다익스트라 알고리즘이다. 사실 단일 시작점 최단 경로 알고리즘으로서는 다익스트라 알고리즘 이외에도
-> 가중치가 음수일 때 적용 가능한 벨만-포드 알고리즘
-> 모든 정점 쌍 사이의 최단 경로를 모두 구하니깐 시간 복잡도가 O(V^3)이 나와버리는 플로이드-워샬 알고리즘이 존재한다. 여기서 다익스트라 알고리즘이 어떻게 돌아가는지에 대해 공부를 해보도록 하겠다.

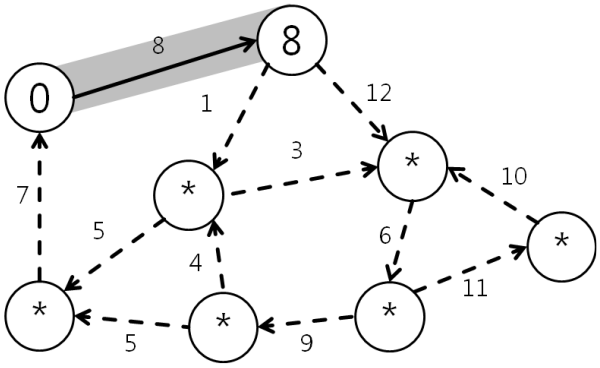
3-1) 다익스트라 알고리즘의 원리
우리가 어느 한 정점에서 시작을 해서 각각 모든 정점으로 이동하는데 있어서 최소의 가중치로 경로를 꾸려 나가는 알고리즘이 바로 다익스트라 알고리즘의 원리이다. 그래서 아래와 같은 그래프를 통해 시작정점을 A로 잡아서 어떤 원리로 돌아가는지에 대하여 공부해보도록 하자.



0단계) 모든 정점의 방문 기록을 무한대 값으로 저장해둔다. Java에선 int형 최댓값인 MAX_VALUE를 저장하는 방법이 있기 마련이다. 여기서 정점 A는 시작점이니 0으로 저장을 해두도록 하자. 왜냐면 A에서 A까지는

A	B	C	D	E	F	G	H
0	∞	∞	∞	∞	∞	∞	∞

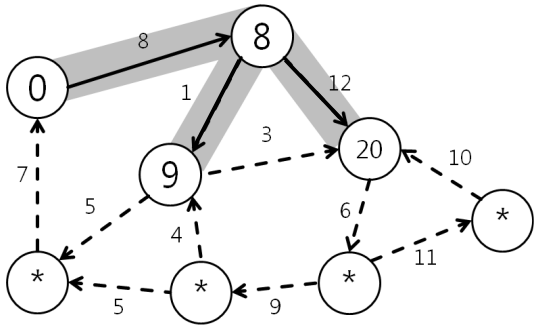
1단계) A가 현재 진출할 수 있는 정점에 대해 파악한다.



현재 A에서 진출할 수 있는 간선으로는 C로 진출이 가능하다. 그러면 A에서 C까지 최소로 이동이 가능한 경로는 8이 되어 기록이 된다.

A	B	C	D	E	F	G	H
0	∞	8	∞	∞	∞	∞	∞

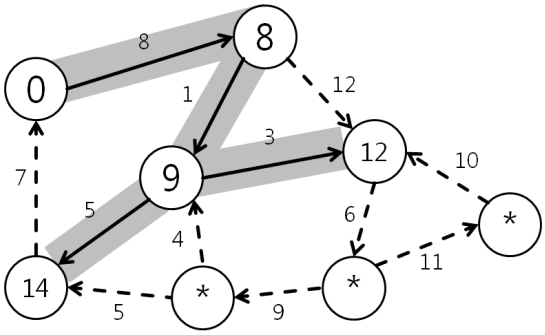
2단계) C에서 일단 진출이 가능한 정점들에 대하여 조사한다.



일단 C에서 진출이 가능한 정점으로는 B, D 정점이다. 그러면 현재 A에서 C까지 경로가 8이니 B, D까지 갈 수 있는 경로는 9랑 12이다. 근데 B를 경유해서 D로 가면 12란 최소 가중치로 이동이 가능한데 이거 요약정리 만든 사람이 잘 못했네 라고 생각을 하는 사람들이 있는데 이는 B, D에서 인접한 정점을 통해서 고려를 하면 되니깐 잘 못 된 건 없으니 안심해도 좋다.

A	B	C	D	E	F	G	H
0	9	8	20	∞	∞	∞	∞

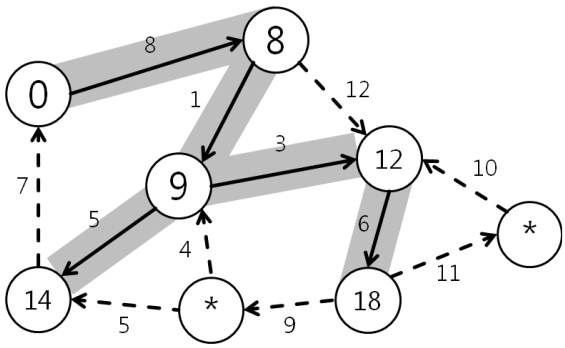
3단계) B에서 진출이 가능한 정점들에 대해 조사한다.



B에서 진출이 가능한 정점은 D랑 E가 있다. 그렇지만 이미 D는 방문을 했음에도 불구하고 최소 거리를 판단하기 위해서 B에서 진출이 가능한 정점들에 대한 거리로 계산을 하여 다시 갱신을 하게 된다. 그래서 D는 20에서 12로 바뀌게 된다. 고로 E는 최초로 방문을 하였으니 14가 나오게 된다.

A	B	C	D	E	F	G	H
0	9	8	12	14	∞	∞	∞

4단계) D에서 진출이 가능한 정점들에 대해 조사한다.



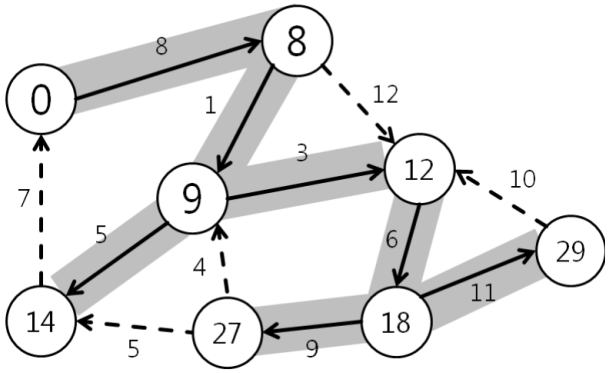
뭔가 가까운 정점들에 대한 조사를 한다는 의미로 생각을 해본다면 방금 전에 Prim 알고리즘을 공부해봤으면 알겠는데 인접 리스트에 저장된 순서대로 탐색

을 하는 과정은 어디가나 똑같다고 생각을 하면 되겠다. 여기서 D에서 진출이 가능한 정점은 바로 G밖에 없다. 그래서 G에 대해 방문 처리를 하고 난 후에 A에서 G까지 거리를 18로 저장을 해둔다.

A	B	C	D	E	F	G	H
0	9	8	12	14	∞	18	∞

그리고 E에 대한 진출이 가능한 정점들에 대해서 계산하면 A로 볼 수 있는데 이미 A는 초기에 0이란 distance가 저장이 되어 있기 때문에 순환이란 무용지물을 맛보게 되고 끝나게 된다.

5단계) G에서 진출이 가능한 정점들에 대해 조사한다.



G에서 진출이 가능한 정점들로는 F랑 H가 되겠다. 이제 모든 정점들에 대해서 최종적으로 방문이 끝났다고 생각할 수 있겠지만, 진짜 만의 하나로 B->D, D->G, G->F, F->E 각각의 가중치가 1인 경우를 생각해보면 F에서 E로 연결을 해서 최솟값을 비교하는 경우가 있다. 그렇지만 여기서는 모든 정점을 방문 완료를 하고 다익스트라 알고리즘을 마무리 하는 시점으로 끝내면 되겠다.

A	B	C	D	E	F	G	H
0	9	8	12	14	27	18	29

3-2) 다익스트라 알고리즘 이해하기
이제 본격적으로 이 알고리즘에 대해서 이해를 해보도록 하자. 좀 보면 Prim 알고리즘보다 똑같다면서 난해하게 느낄 수도 있겠지만 방금 전에 과정들을 생각해서 공부해보면 도움 되겠다.

```
static Vertex extractMin(Vertex[] V, HashSet<Vertex>
visited, HashMap<Vertex, Integer> distance) {
  int min = Integer.MAX_VALUE;
  Vertex vertex = null;
  for (Vertex v : V) {
    if (visited.contains(v) == false && distance.get(v) <
min) {
      vertex = v;
      min = distance.get(v);
    }
  }
  return vertex;
}

static void dijkstra(Vertex[] V, Vertex start,
HashMap<Vertex, Integer> distance,
HashMap<Vertex, Vertex> previous) {
  HashSet<Vertex> visited = new HashSet<>();
  for (Vertex v : V)
    distance.put(v,Integer.MAX_VALUE);
  distance.put(start, 0); // 1
  while (visited.size() < V.length) {
    Vertex u = extractMin(V, visited, distance);
    visited.add(u); // 2
      for (Edge e : u.edge) {
        Vertex v = e.vertex;
        if (visited.contains(v) == false &&
        distance.get(v) > distance.get(u) + e.weight) {
          distance.put(v, distance.get(u) + e.weight);
          previous.put(v, u); // 3
        }
    }
  }
}
```

1 : 아까 0단계와 똑같이 처음에 시작하는 정점에 대해서만 0으로 해두고 나머지는 무한대로 지정을 해두는 원리로 생각하면 되겠다.
2 : 여기서 extractMin 함수가 무엇일까? 실제로 정점에 대한 방문은 진입을 한 경우의 시점이 아니라 진입을 하고 난 후에 진출을 하는 시점에서 바뀔 때가 바로 정식적으로 방문 처리가 되는 것이다. 쉽게 이야기 하면 방문을 완료한 정점에 대해 최솟값을 찾아서 효율적으로 경로를 찾기 위한 작업으로 생각하면 되겠다.
3 :여기서 distance는 시작 정점에서 각 정점들에 대해 도달을 하는 최솟값들을 저장하는 역할을 한다고 생각을 하면 되겠다. 그리고 previous는 최소로 나오는 경로에 대해서 도달점 바로 이전의 정점으로 생각을 하면 되겠다. 이를 작성한 이유는 실제로 시작된 정점에서부터 어느 정점을 마지막 이전에 경유했는지에 대해 판단을 하면 최단 경로에 대해 효율적으로 생각을 해볼 수 있기 때문에 쓴 것으로 판단된다.

3-3) 다익스트라 알고리즘의 시간 복잡도
다익스트라 알고리즘의 최종 시간 복잡도는 각 정점들(O(V))에 대해서 진출할 수 있는 간선들에 대해 distance 값들을 선형적으로 정리(O(E))하는데 아까와 같은 알고리즘의 시간 복잡도는 O(VE)로 볼 수 있다. 그렇지만 힙 자료구조를 이용한다면 어떻게 달라질까? distance 값들을 선형적으로 정리하는데 있어서 시간 복잡도가 O(log V)으로 줄어드는 현상을 맛볼 수 있다. 아래의 코드를 살펴보도록 하자.

```
static void dijkstra(Vertex[] V, Vertex start,
HashMap<Vertex, Vertex> previous) {
    VertexHeap heap = new VertexHeap(V); // *
    start.distance = 0;
    heap.heapifyUp(start.index); // *
    while (heap.size() > 0) {
      Vertex u = heap.extractMin(); // *
      u.visited = true;
      for (Edge e : u.edge) {
        Vertex v = e.vertex;
        if (v.visited == false && v.distance >
u.distance + e.weight) {
          v.distance = u.distance + e.weight;
          heap.heapifyUp(v.index);
          previous.put(v, u);
        }
      }
```

```
  }
}
```

여기서 주목해야 하는 점은 바로 *이다. 모든 간선들에 대해서 Heap 자료구조에 저장을 하고 난 후에 heapifyUp을 통해 heap을 정리하면서 이용하면 시간복잡도가 O(log V)로 줄어들게 된다. 그 Heap이 어떻게 구성되어 있는지에 대해 잠깐 살펴보도록 하자.

```
public class VertexHeap {
  Vertex[] a;
  int end;
  public VertexHeap(Vertex[] a) {
    this.a = a.clone();
    this.end = a.length - 1;
    buildHeap();
    for (int i = 0; i < a.length; ++i) {
      a[i].distance = Integer.MAX_VALUE;
      a[i].index = i;
    }
  }
  public Vertex extractMin() {
    Vertex min = a[0];
    a[0] = a[end];
    --end;
    heapifyDown(0);
    return min;
  }
  public int size() {
    return end + 1;
  }
  public void swap(int i, int j) {
    Vertex temp = a[i];
    a[i] = a[j];
    a[j] = temp;
    a[i].index = i;
    a[j].index = j;
  }
  public void buildHeap() {
    for (int i = end / 2; i >= 0; --i)
      heapifyDown(i);
  }
  public void heapifyDown(int k) {
    int left = 2 * k, right = 2 * k + 1;
    int smaller;
    if (right <= end)
      smaller = (a[left].compareTo(a[right]) < 0) ?
left : right;
    else if (left <= end) smaller = left;
    else return;
    if (a[smaller].compareTo(a[k]) < 0) {
      swap(smaller, k);
      heapifyDown(smaller);
    }
  }
  public void heapifyUp(int k) {
    while (k > 0) {
      int parent = (k - 1) / 2;
      if (a[parent].compareTo(a[k]) < 0) break;
        swap(parent, k);
        k = parent;
      }
  }
}
```

일단 a의 요소는 당연히 그래프에 만들어진 정점들에 대해서 복제를 해오고 난 후에 이용을 하고, end는 a의 마지막 인덱스를 저장을 해서 정점끼리 distance(여기서는 정점이란 객체의 인스턴스에 포함되어 있다.)를 비교를 하고 난 후에 Heap 자료구조를 통해(여기서는 최소 Heap을 이용) 정리를 해서 최솟값을 골라내면서 이용하면 되는데 이 메소드는 extractMin() 메소드가 책임지게 된다. 이외의 다익스트라 알고리즘은 크게 달라지는 것이 없기 때문에 오로지 시간 복잡도를 줄여준다는 역할을 한다고 생각하면 되겠다. 참고로 아래는 Vertex 정점에 어떤 인스턴스가 추가되었는지 비교하기 위해서 아래 클래스를 참고하도록 하자.

```
public class Vertex implements
Comparable<Vertex> {
    String title;
    Edge[] edge;
    int distance, index;
    boolean visited;

    public Vertex(String title) {
        this.title = title;
    }

    @Override
    public int compareTo(Vertex v) {
        return this.distance - v.distance;
    }
}
```